지우는 기술

기능과 코드를 안전하게 없애는 엔지니어의 규율

ThakiCloud (Hyojung Han)

지우는 기술

기능과 코드를 안전하게 없애는 엔지니어의 규율

ThakiCloud (Hyojung Han)

Contents

1 지우지 못하는 조직의 해부

새 기능을 만드는 법은 어디서나 배웁니다. 설계 문서를 쓰고, 테스트를 붙이고, 배포 파이프라인을 태우는 절차는 신입 온보딩 첫 주에 나옵니다. 반대 방향을 가르치는 곳은 드물죠. 무엇을 없애야 하는지, 없애도 되는지 어떻게 아는지, 없애다가 사고가 나면 어디까지 되돌릴 수 있는지는 대부분 각자 감으로 처리합니다. 그 결과 제품은 한 방향으로만 자랍니다.

이 책은 그 반대 방향을 다룹니다. 첫 장에서는 기술이 아니라 진단부터 합니다. 왜 우리 팀은 지우지 못할까요.

1.1 코드 한 줄이 실제로 청구하는 비용

"이 코드는 그냥 거기 있을 뿐인데 무슨 비용이 드나"라는 반문을 자주 듣습니다. 디스크는 싸고 CPU도 남으니 틀린 말처럼 들리지 않습니다. 문제는 청구서가 인프라 요금이 아니라 사람의 시간으로 날아온다는 점입니다.

살아 있는 코드 경로 하나는 매달 다음을 청구합니다.

- 읽기 비용. 근처를 수정하려는 사람이 이 코드가 무엇인지 파악하는 데 쓰는 시간입니다. 아무도 쓰지 않는 기능이라도 옆 함수를 고치려면 읽어야 합니다.
- 테스트 시간. 파이프라인이 매번 이 경로를 빌드하고 검증합니다. 팀 전체의 대기 시간에 비례해 곱해집니다.
- 사고 표면. 인증, 권한, 직렬화, 외부 호출이 걸린 모든 경로는 취약점 공지가 나올 때마다 점검 대상이 됩니다.
- 마이그레이션 세금. 프레임워크를 올리거나 데이터베이스를 옮길 때, 죽은 기능도 똑같이 이사 비용을 받습니다. 이 세금이 제일 비쌉니다.

- 인지 부하. 온보딩하는 사람이 “이건 왜 있어요”라고 묻고, 아무도 확실히 대답하지 못하는 상황이 반복됩니다.

앞의 네 가지는 측정할 수 있고, 마지막 하나는 측정하기 어렵지만 이직 면담에서 자주 등장합니다. 이 비용은 일회성이 아니라 매달 자동 갱신된다는 사실이 중요합니다. 기능을 만들 때는 한 번 결제하고, 유지할 때는 구독료를 냅니다.

1.2 삭제가 매번 뒤로 밀리는 네 가지 이유

우선순위 회의에서 폐기 작업이 항상 지는 이유는 게으름이 아닙니다. 구조가 그렇게 짜여 있습니다.

가장 먼저, 보상이 비대칭입니다. 기능을 출시하면 데모가 있고 릴리스 노트가 있고 축하가 있습니다. 기능을 없애면 화면에서 뭔가 사라진 것뿐이라 자랑할 자리가 없습니다. 성과 평가에 “제거한 코드”를 적는 칸이 있는 팀은 흔치 않죠.

증거도 없습니다. “이거 쓰는 사람 있나요”라는 질문에 확신을 갖고 답할 계측이 대개 없습니다. 없으면 보수적으로 남기게 되고, 남기면 계측할 이유도 사라집니다. 이 고리가 몇 년 돌면 아무도 진실을 모르는 기능이 수십 개 쌓입니다.

책임 구조도 비대칭입니다. 기능을 방치해서 생긴 유지비는 팀 전체가 조금씩 나눠 냅니다. 반면 삭제해서 사고가 나면 삭제한 사람 한 명의 이름이 사후 보고서에 남습니다. 합리적인 개인은 방치를 택합니다.

마지막으로, 되돌리기가 두렵습니다. 정확히는 되돌리는 절차를 연습해 본 적이 없어서 두렵습니다. 복구 훈련을 한 번이라도 해 본 팀은 삭제에 훨씬 과감합니다. 두려움의 크기는 리스크가 아니라 연습량에 반비례합니다.

1.3 유령 기능 3분류

폐기 후보를 한 덩어리로 다루면 진도가 안 나갑니다. 성질이 다른 것을 섞어 놓았기 때문입니다. 세 가지로 나누면 첫 조치가 명확해집니다.

유형	관찰되는 신호	첫 조치
화석	호출 0, 참조 0	바로 격리
좀비	호출은 있음	호출자 정체 확인
인질	소수가 강하게 씀	대안 협상

화석은 쉽습니다. 어떤 진입점에서도 도달하지 않는 코드, 라우팅 테이블에서 빠진 화면, 참조가 끊긴 설정입니다. 정적 분석으로 상당수 잡히고, 지워도 사용자가 알아채지 못합니다. 문제는 화석이 전체의 일부라는 점이죠.

좀비가 진짜 골칫거리입니다. 대시보드에는 호출 수가 찍히는데 그 호출이 누구 것인지 아무도 모릅니다. 헬스체크일 수도 있고, 5년 전 만든 배치가 스스로를 부르는 것일 수도 있고, 실제 고객일 수도 있습니다. 좀비는 호출량이 아니라 호출자를 밝혀야 판정이 됩니다. 다음 장 전체가 이 문제를 다룹니다.

인질은 기술 문제가 아니라 협상 문제입니다. 전체의 0.3퍼센트가 쓰지만 그 0.3퍼센트가 매출의 큰 몫을 차지하는 기능이 여기 해당합니다. 여기서 필요한 건 더 정교한 코드 분석이 아니라 대체 경로, 이전 지원, 그리고 종료 날짜에 대한 명확한 약속입니다.

1.4 유지 원가를 숫자로 바꾸기

경영진에게 "코드가 지저분합니다"라고 말하면 우선순위가 오르지 않습니다. "이 모듈이 분기당 사람 시간 얼마를 먹습니다"라고 말하면

대화가 달라집니다. 완벽한 숫자는 필요 없고, 반박 가능한 근사치면 충분합니다.

간단한 추정식을 씁니다.

```
연간 유지 원가(사람일)
 = 연간 변경 횟수 x 회당 이해 및 리뷰 시간
 + 연간 인시던트 시간 x 이 모듈의 지분
 + 연간 파이프라인 실행 시간 x 이 모듈 비중
 + 예정된 대형 마이그레이션 x 이 모듈 이전 비용
```

숫자는 버전 관리 이력, 인시던트 기록, 파이프라인 로그에서 대부분 뽑을 수 있습니다. 정밀도가 아니라 순서가 목적입니다. 후보 20개를 이 식에 넣으면 상위 3개가 나머지 17개를 합친 것보다 비싼 경우가 흔합니다. 그 3개만 제대로 지워도 그해 폐기 작업의 목표는 달성됩니다.

여기에 하나를 곱해 줍니다. 되돌리기 난이도입니다. 값이 낮을수록 먼저 손대기 좋습니다.

- 코드 경로 제거: 되돌리기 쉬움. 커밋 되돌리면 끝.
- 설정과 플래그 제거: 중간. 재배포 필요.
- 데이터 삭제: 어려움. 백업이 없으면 복구 불가.
- 외부 공개 계약 종료: 가장 어려움. 상대의 일정에 묶입니다.

유지 원가가 크고 되돌리기가 쉬운 것부터 처리합니다. 반대로 유지 원가가 작은데 되돌리기가 어려운 항목은 굳이 건드리지 마세요. 그건 폐기 작업이 아니라 위험 취미입니다.

1.5 현장 스케치: 정산 화면 세 개

한 결제 플랫폼의 관리자 콘솔에 정산 화면이 세 개 있었습니다. 이름은 정산, 신정산, 정산v2였습니다. 순서대로 만들어진 것도 아니었고, 어느 것이 최신인지 아는 사람도 남아 있지 않았습니다. 세 화면은 같은 테이블을

조금씩 다른 방식으로 읽었고, 그중 하나만 새 수수료 정책을 반영하고 있었습니다.

팀은 반년 동안 이 상태를 알고 있었지만 손대지 않았습니다. 이유는 앞에서 말한 그대로였습니다. 어느 화면을 누가 쓰는지 증거가 없었고, 잘못 지우면 월말 정산이 틀어질 수 있었으며, 정리해 봐야 릴리스 노트에 쓸 문장이 없었습니다.

전환점은 기술적 발견이 아니라 계산이었습니다. 세 화면을 유지하느라 수수료 정책이 바뀔 때마다 같은 로직을 세 번 고치고 세 번 검증하고 있었고, 지난 1년간 그런 변경이 아홉 번 있었습니다. 회당 이해와 리뷰에 대략 이틀이 들었으니 스무 사람일 남짓이 여기에 묻힌 셈입니다 [추정]. 여기에 정산 관련 인시던트 두 건 중 한 건이 "어느 화면 기준으로 본 숫자인가"에서 시작됐다는 기록이 붙었습니다.

숫자가 나오자 논의가 짧아졌습니다. 정리 작업이 다음 분기 계획에 들어갔고, 실제로는 지우기 전에 계측을 먼저 붙이는 일부터 시작했습니다. 두 달 뒤 화면 두 개가 사라졌고, 수수료 정책 변경 작업은 이틀에서 반나절로 줄었습니다.

이 사례에서 눈여겨볼 부분은 기술이 아닙니다. 팀이 이미 알고 있던 사실을 다른 언어로 옮긴 것뿐입니다. "지저분하다"는 취향의 진술이라 반박당하지만, "분기마다 사람일 다섯이 여기로 샙니다"는 예산의 진술이라 다음 행동을 부릅니다.

1.6 지우기 좋은 순간과 나쁜 순간

같은 삭제라도 시점에 따라 위험이 크게 달라집니다. 좋은 시점부터 봅니다.

- 그 영역을 어차피 크게 손봐야 할 때. 마이그레이션이나 리팩터링을 하면서 함께 걷어내면 검증 비용을 나눠 씁니다.

- 담당자가 아직 팀에 있을 때. 원래 만든 사람이 남아 있는 동안이 판정 비용이 가장 쌉니다.
- 트래픽이 낮고 관측이 촘촘한 시기. 새벽 배포가 아니라 사람이 많이 보고 있는 평일 오전이 더 안전합니다.

나쁜 시점도 분명합니다. 분기 마감 주간, 대형 프로모션 직전, 감사 대응 기간, 핵심 담당자 휴가 중에는 아무리 확실해 보여도 미룹니다. 폐기는 급한 일이 아니라는 점이 오히려 장점입니다. 언제 할지 우리가 고를 수 있으니까요.

1.7 이 장의 결과물

첫 장을 마치며 팀이 손에 쥐어야 할 것은 문서 한 장입니다. 후보 목록, 각 후보의 유형, 대략적인 연간 원가, 되돌리기 난이도. 이 표가 있으면 다음 회의에서 "언젠가 정리해야죠"라는 말이 "이번 분기에 이 두 개"로 바뀝니다.

아직 아무것도 지우지 않았다는 점을 기억하세요. 판정 없이 지우는 건 청소가 아니라 사고입니다. 무엇이 정말 안 쓰이는지 증명하는 방법이 다음 장입니다.

2 증거로 판정하기: 안 쓰인다는 말의 기준

폐기 작업이 실패하는 가장 흔한 방식은 잘못 지우는 것이 아니라 영원히 논쟁하는 것입니다. “이거 안 쓰지 않아요”와 “혹시 모르니 두죠”가 무한히 반복됩니다. 두 진술 모두 증거가 없어서 결론이 나지 않습니다.

해법은 더 똑똑한 토론이 아니라 증거 등급입니다. 어떤 근거가 어느 등급인지 미리 정해 두고, 등급별로 허용되는 조치를 고정합니다. 그러면 회의는 “지울까 말까”가 아니라 “지금 몇 등급인가”를 확인하는 자리가 됩니다. 훨씬 짧게 끝납니다.

2.1 사용 증거 등급 E0에서 E3

등급	근거	허용 조치
E0	감과 기억뿐	계측 부착만
E1	정적 참조 없음	격리와 예고
E2	런타임 무호출	차단 실험
E3	호출자 전수 확인	제거

E0은 아무 증거도 없는 상태입니다. 여기서 할 수 있는 유일한 행동은 계측을 붙이는 것입니다. 이 규칙 하나만 지켜도 사고의 절반은 사라집니다. 감으로 지우지 않겠다는 약속이니까요.

E1은 코드베이스 안에서 아무도 이 경로를 참조하지 않는다는 확인입니다. 정적 분석, 라우팅 테이블, 빌드 산출물에서 나옵니다. 강력하지만 반쪽입니다. 리플렉션, 문자열로 조립되는 경로, 설정 파일에 적힌 클래스 이름, 외부에서 직접 때리는 엔드포인트는 정적 분석에 잡히지 않습니다.

E2는 실제 운영 환경에서 일정 기간 호출이 없었다는 관측입니다. E1과 E2를 함께 확보하면 대부분의 내부 기능은 안전하게 다음 단계로 갑니다.

E3은 호출이 있는 경우에 필요한 등급입니다. 남은 호출자가 누구인지 전부 식별했고, 각각에 대해 대안이나 종료 합의가 있는 상태입니다. 외부 공개 API를 없앨 때는 여기까지 와야 합니다.

2.2 3층 탐지: 정적, 런타임, 계약

증거는 세 곳에서 나옵니다. 한 층만 보면 반드시 틀립니다.

정적 층은 소스 안의 참조 관계입니다. 죽은 심벌, 도달 불가 분기, 아무도 임포트하지 않는 모듈을 찾습니다. 값이 싸고 빠르니 먼저 돌립니다. 단, 앞서 말한 동적 참조는 놓친다는 점을 전제로 다룹니다. 문자열로 조립되는 이름은 전체 검색으로 한 번 더 훑는 습관을 들이세요.

런타임 층은 실행 중 관측입니다. 여기서 핵심은 로그가 아니라 카운터입니다. 로그는 보존 기간이 짧고 샘플링되며 비쌉니다. 기능 단위 카운터는 싸고 오래 남습니다. 다음 원칙을 지킵니다.

- 이름은 코드 위치가 아니라 기능 단위로 붙입니다. 함수명을 쓰면 리팩터링 때 끊깁니다.
- 라벨은 최소로 유지합니다. 사용자 식별자를 라벨에 넣으면 카디널리티가 터집니다.
- 대신 액터 유형을 구분합니다. 내부 배치, 헬스체크, 모니터링, 직원, 외부 고객 정도면 충분합니다.
- 카운터를 붙인 날짜를 기록합니다. 관측 창의 시작점이 언제인지 나중에 반드시 필요합니다.

계약 층은 우리 시스템 밖에서 오는 사용입니다. 공개 API, 웹훅, 이벤트 토픽, 공유 데이터베이스 뷰, 남이 스크래핑하는 화면이 여기 속합니다. 이 층은 코드로 찾을 수 없고 접근 기록으로 찾습니다. 인증 토큰별 호출량,

클라이언트 식별 헤더, 게이트웨이 로그를 봅니다. 계약 층이 비어 있다는 걸 증명하지 못하면 E3에 도달할 수 없습니다.

2.3 관측 창은 달력이 아니라 주기로 잡는다

"2주 동안 호출이 없었으니 지웁시다"는 실무에서 가장 위험한 문장입니다. 소프트웨어는 달력이 아니라 업무 주기를 따라 쓰입니다.

최소한 다음을 지나 보내야 합니다.

- 월 단위 마감과 정산
- 분기 마감과 실적 보고
- 연 1회 이벤트, 예를 들어 연말정산, 감사, 갱신 시즌
- 대형 프로모션이나 계절 성수기

연 1회 배치를 11개월째에 지우면 남은 한 달 뒤에 사고가 납니다. 그래서 관측 창의 기본값을 한 번의 연간 주기로 잡되, 기능의 성격에 따라 줄입니다. 실험용 플래그는 2주면 충분하고, 사용자 화면은 한 분기, 회계나 규제와 닿은 경로는 1년입니다. 이 값을 팀 규칙으로 적어 두면 매번 다시 논쟁하지 않아도 됩니다.

관측 창을 기다리는 동안 아무것도 안 하는 건 아닙니다. 나중에 나올 예고와 마찰 단계를 병행할 수 있습니다.

2.4 위양성을 만드는 트래픽 걷어내기

호출 수가 0이 아니라고 해서 살아 있는 건 아닙니다. 다음은 실무에서 자주 발견되는 가짜 사용입니다.

- 헬스체크와 가용성 모니터가 주기적으로 때리는 요청
- 통합 테스트와 스테이징 트래픽이 운영 지표에 섞인 경우
- 우리 시스템의 다른 부분이 자기 자신을 호출하는 내부 루프

- 검색 엔진 크롤러와 링크 미리보기 봇
- 오래전 만든 대시보드가 30초마다 갱신하는 조회
- 재시도 폭풍. 실패한 호출 한 번이 지수적으로 부풀려진 경우

이걸 걸러내면 호출 수가 하루 수천 건에서 실제 사용자 두 명으로 줄어드는 일이 흔합니다. 반대 방향의 오류도 있습니다. 샘플링된 추적만 보고 "거의 없다"고 판정했는데, 샘플링에서 빠진 소수 경로가 매출의 핵심이었던 경우죠. 그래서 판정용 계측은 샘플링하지 않는 것을 원칙으로 합니다. 카운터 하나 늘리는 비용은 무시할 만합니다.

2.5 폐기 판정 카드

증거가 모이면 한 장으로 정리합니다. 길게 쓰면 아무도 안 읽으니 다음 항목만 채웁니다.

```
대상: 관리자 콘솔 정산v1 화면
유형: 좀비
증거 등급: E2 (2026-03-01 계측 부착, 관측 90일)
관측 결과: 총 호출 412건, 그중 사내 모니터 400건,
 직원 계정 12건, 외부 고객 0건
남은 호출자: 운영팀 2명 (대체 화면 정산v2 존재)
되돌리기 난이도: 중 (재배포 필요)
연간 유지 원가: 사람일 12 [추정]
제안 조치: 4월 예고, 5월 차단 실험, 6월 제거
철회 기준: 차단 중 고객 문의 1건이라도 발생 시 즉시 복구
```

이 카드의 진짜 가치는 마지막 두 줄입니다. 언제 되돌릴지를 미리 적어 두면, 실제로 문제가 생겼을 때 논쟁 없이 손이 움직입니다. 사고 대응에서 가장 비싼 것은 장애 자체가 아니라 "이게 롤백할 상황인가"를 판단하는 데 쓰는 20분입니다.

2.6 계측이 없는 시스템에서 시작하기

지금까지의 이야기는 계측이 있다는 전제 위에 서 있습니다. 현실의 레거시는 대개 그렇지 않죠. 아무 지표도 없는 10년짜리 시스템 앞에서 무엇부터 해야 할까요.

가장 싼 순서로 붙입니다. 애플리케이션 코드를 건드리지 않고 얻을 수 있는 것부터 씁니다.

- 웹 서버와 게이트웨이의 접근 기록. 이미 남고 있는 경우가 많습니다. 경로별 집계만 뽑아도 화면과 엔드포인트의 대략적 순위가 나옵니다.
- 데이터베이스의 쿼리 통계. 어떤 테이블이 최근에 읽혔는지, 어떤 인덱스가 한 번도 안 쓰였는지 알려 줍니다. 테이블 폐기 판단에서 특히 유용합니다.
- 배치 스케줄러의 실행 기록. 등록은 되어 있으나 몇 년째 실패만 하고 있는 작업이 종종 발견됩니다.
- 오류 추적 도구. 오류가 한 번도 안 난 경로는 안 쓰이는 경로일 가능성이 큽니다. 확정 근거는 아니지만 후보 목록을 만드는 데는 충분합니다.

여기까지는 코드 변경이 없어 반나절이면 끝납니다. 그다음에 진짜 카운터를 붙입니다. 전부 한 번에 붙이려 하지 말고, 앞 장에서 만든 상위 후보 다섯 개에만 답니다. 계측 자체도 부채가 될 수 있으니 판정이 끝나면 카운터도 같이 걷어냅니다.

2.7 사람에게 묻는 법

계측이 답하지 못하는 질문이 남습니다. “이 기능이 없으면 무슨 일이 벌어지나요”는 로그가 대답하지 못합니다.

설문을 돌리는 방식은 잘 안 먹힙니다. 응답률이 낮고, 응답한 사람은

대체로 모든 기능이 필요하다고 답합니다. 사람은 잃을 가능성 앞에서 실제 사용량보다 크게 대답합니다.

더 잘 작동하는 질문은 과거형입니다. "이 기능을 마지막으로 쓴 게 언제인가요"나 "그때 무엇을 하려고 쓰셨나요"처럼 구체적인 사건을 묻습니다. 답이 안 나오면 실질적으로 안 쓰는 겁니다. 영업이나 고객 지원 담당자에게는 "최근 6개월간 이 기능 때문에 들어온 문의가 있나요"를 묻습니다. 기억이 아니라 티켓 기록을 근거로 답할 수 있는 질문이라 훨씬 정확합니다.

그리고 가장 정직한 질문 방식이 하나 더 있습니다. 잠깐 꺼 보는 것입니다. 그게 다음 장의 주제입니다.

2.8 판정이 안 나올 때

증거를 모아도 결론이 안 나는 경우가 있습니다. 이럴 때 쓸 수 있는 처방은 세 가지입니다.

범위부터 쪼갭니다. 기능 전체는 애매해도 그 안의 특정 옵션이나 화면 하나는 명백히 화석인 경우가 많습니다. 큰 덩어리를 못 지우면 가장자리부터 깎습니다.

소유자를 지정하고 기한을 거는 방법도 있습니다. "6월까지 이 기능의 유지 근거를 대는 사람이 없으면 폐기 대상"이라고 공지합니다. 소유자 없는 기능은 이미 죽은 기능입니다.

그래도 안 되면 판정을 미루되 비용은 줄입니다. 지우지 못한다면 최소한 테스트 범위에서 제외하거나, 별도 모듈로 격리하거나, 새 기능이 이 코드에 의존하지 못하게 막습니다. 완전한 삭제가 유일한 승리는 아닙니다.

증거가 갖춰졌다면 이제 손을 댈 차례입니다. 다음 장은 순서 이야기입니다.

3 안전한 철거 순서: 유예에서 제거까지

증거가 모였다고 바로 지우면 안 됩니다. 폐기 사고의 대부분은 판정이 틀려서가 아니라 순서가 틀려서 납니다. 코드를 먼저 지우고 데이터를 나중에 지우면 되돌릴 수 없는 상태에 먼저 도달합니다. 예고 없이 차단하면 기술적으로는 완벽한 작업이 신뢰 사고가 됩니다.

이 장은 순서만 다룹니다. 다섯 단계를 지키면 웬만한 폐기는 지루하게 끝납니다. 폐기 작업의 목표는 지루함입니다.

3.1 다섯 단계와 종료 조건

단계	하는 일	다음으로 넘어가는 조건
1 관측	계측 부착	증거 등급 E2 도달
2 예고	종료일 통보	이의 처리 완료
3 마찰	경고와 지연 추가	잔여 사용 감소 확인
4 차단	기능을 끔	소등 기간 무사고
5 제거	코드와 자원 삭제	복구 창 만료

각 단계의 핵심은 다음 단계로 넘어가는 조건입니다. 조건 없이 일정만 있으면 결국 일정에 밀려 조건을 건너뜁니다.

3.2 1단계 관측

앞 장의 내용이 여기 해당합니다. 한 가지만 덧붙이면, 이 단계에서 이미 되돌리기 계획을 씁니다. 나중에 쓰려고 미루면 정작 급할 때 없습니다.

어떤 커밋을 되돌리면 되는지, 어떤 플래그를 켜면 되는지, 데이터 백업이 어디 있는지를 카드에 적어 둡니다.

3.3 2단계 예고

내부 기능이든 외부 API든 예고는 생략하지 않습니다. 예고문에는 네 가지만 있으면 됩니다. 무엇이 언제 사라지는지, 대신 무엇을 쓰면 되는지, 문제가 있으면 누구에게 말하면 되는지, 그리고 아무 조치도 안 하면 어떻게 되는지입니다.

예고 기간은 상대에 따라 다릅니다.

- 내부 팀만 쓰는 기능: 2주에서 한 달
- 사내 전체가 쓰는 도구: 한 분기
- 외부 고객이 붙은 API: 최소 두 분기, 계약이 있으면 계약 조건 우선

예고를 문서에만 남기면 아무도 읽지 않습니다. 실제 사용자가 그 기능을 쓰는 자리에 배치해야 합니다. 화면이면 화면 안 배너, API면 응답 헤더, 라이브러리면 경고 로그입니다. 예고는 공지가 아니라 접촉입니다.

여기서 이의가 들어옵니다. 이의는 좋은 신호입니다. 우리가 몰랐던 사용자를 공짜로 찾아 준 셈이니까요. 이의를 처리한 결과가 항상 일정 연기일 필요는 없습니다. 대체 경로 안내, 데이터 내보내기 지원, 개별 유예 기간 부여도 정당한 처리입니다.

3.4 3단계 마찰

예고했는데도 사용량이 줄지 않는 일이 자주 있습니다. 사람은 마감 직전에 움직이니까요. 그래서 마찰을 넣습니다. 사용을 막지는 않되 불편하게 만들어 이전을 앞당깁니다.

- 화면 진입 시 종료 안내 배너를 확인 클릭으로 바꿉니다.

- API 응답에 종료 헤더를 추가하고, 남은 날짜를 숫자로 넣습니다.
- 응답에 인위적 지연을 조금 넣습니다. 100밀리초 정도면 자동화된 호출자에게 신호가 됩니다.
- 하루 호출 한도를 걸어 점진적으로 낮춥니다.

마찰 단계의 목적은 처벌이 아니라 발견입니다. 지연이나 한도에 반응해서 연락해 오는 사람이 바로 진짜 사용자입니다. 조용히 넘어간 호출자는 대체로 자동화이거나 관심 없는 사용입니다. 마찰은 사용자 목록을 정확하게 만드는 저렴한 도구입니다.

주의할 점도 있습니다. 결제, 로그인, 안전과 관련된 경로에는 인위적 지연을 넣지 않습니다. 여기서는 배너와 헤더까지만 씁니다.

3.5 4단계 차단, 즉 소등 훈련

차단은 코드를 지우지 않고 기능만 끄는 단계입니다. 스위치 하나로 되돌릴 수 있어야 합니다. 이 단계가 폐기 작업의 진짜 시험입니다.

한 번에 영구히 끄지 말고 점증합니다.

1. 1시간 소등. 업무 시간 중 사람이 많이 보고 있을 때 합니다.
2. 문제 없으면 하루 소등. 주말은 피합니다. 아무도 안 보는 시간에 성공하는 건 정보가 없습니다.
3. 일주일 소등. 여기서 조용하면 주간 주기의 사용자는 없다는 뜻입니다.
4. 한 달 소등. 월 마감 주기를 지나 보냅니다.

각 소등마다 시작 전에 두 가지를 확정합니다. 어떤 신호가 보이면 즉시 켜는가, 그리고 켜는 데 몇 분이 걸리는가. 첫 번째는 대개 고객 문의, 오류율 상승, 내부 팀의 호출입니다. 두 번째는 훈련으로 줄입니다. 소등 훈련은 삭제 연습이자 복구 연습입니다.

소등 중 문제가 생기면 되돌리는 게 실패가 아니라 이 단계의 목적입니다. 진짜 실패는 되돌릴 수 없는 상태로 만들어 놓고 소등하는 경우입니다.

3.6 5단계 제거

여기서도 순서가 있습니다. 되돌리기 쉬운 것부터 없앱니다. 직관과 반대로 느껴질 수 있는데, 가장 위험한 것을 마지막에 남겨 두어야 그때까지 계속 되돌릴 수 있기 때문입니다.

1. 사용자 진입점. 메뉴, 링크, 라우팅.
2. 프런트엔드 화면과 컴포넌트.
3. 서버 측 처리 로직과 엔드포인트.
4. 배경 작업, 스케줄, 큐 소비자.
5. 설정, 비밀값, 접근 권한.
6. 계측과 대시보드. 이 시점에는 볼 것이 없습니다.
7. 데이터와 스키마. 마지막입니다. 다음 장 전체가 이 이야기입니다.

각 단계 사이에 최소 한 번의 배포 간격을 둡니다. 같은 배포에 전부 몰아넣으면 문제가 생겼을 때 어느 것이 원인인지 알 수 없고, 되돌리기도 통째로 해야 합니다.

3.7 복구 창

제거를 마쳤다고 작업이 끝난 것은 아닙니다. 일정 기간 동안 빠르게 되돌릴 수 있는 상태를 유지합니다. 이것을 복구 창이라 부릅니다.

- 코드는 되돌릴 커밋 해시를 카드에 적어 둡니다.
- 데이터는 백업본의 위치와 복원 절차를 적고, 실제로 복원해 봅니다. 검증하지 않은 백업은 백업이 아닙니다.
- 창의 길이는 관측 창과 같은 논리로 잡습니다. 회계 경로는 한 번의 마감을 지나 보냅니다.

복구 창이 지나면 폐기 카드를 닫습니다. 닫을 때 한 문단을 남깁니다. 무엇을 없앴고, 무엇이 놀라웠고, 다음에 무엇을 다르게 할지. 이 기록이 쌓이면 다음 폐기가 눈에 띄게 빨라집니다.

3.8 스위치가 없는 기능을 끄는 법

4단계는 "스위치 하나로 끈다"를 전제합니다. 그런데 오래된 기능일수록 스위치가 없습니다. 만들 때 플래그라는 개념이 없었거나, 있었더라도 정리하면서 지워졌기 때문입니다.

그래서 폐기 작업의 첫 코드 변경은 삭제가 아니라 추가인 경우가 많습니다. 끄는 장치를 먼저 답니다. 이때 완벽한 기능 플래그 시스템을 도입할 필요는 없습니다. 다음 정도로 충분합니다.

- 설정값 하나로 진입점에서 분기시켜 안내 화면을 보여 줍니다.
- 게이트웨이나 리버스 프록시 수준에서 경로를 막습니다. 애플리케이션을 건드리지 않아 가장 빠릅니다.
- 권한 체계가 있다면 해당 권한을 회수하는 방식도 됩니다. 다만 권한 오류 화면이 사용자에게 어떻게 보이는지 먼저 확인하세요.

여기서 자주 나오는 실수는 스위치를 껐을 때의 사용자 경험을 설계하지 않는 것입니다. 500 오류나 빈 화면이 뜨면 소등 훈련이 장애 신고로 바뀝니다. 꺼진 상태에서 사용자가 보게 될 문구를 미리 준비합니다. "이 기능은 6월 30일부터 제공되지 않습니다. 대신 정산v2 화면을 이용해 주세요"처럼 다음 행동이 적혀 있어야 합니다.

3.9 여러 팀이 걸려 있을 때

폐기 대상이 팀 경계를 넘으면 기술보다 조율이 어렵습니다. 여기서 통하는 방법은 세 가지입니다.

먼저 종료일을 한 번만 정하고 흔들지 않습니다. 한 번 미루면 두 번째 요청이 반드시 옵니다. 미루는 대신 개별 유예를 주는 편이 낫습니다. 전체 일정은 유지하되 특정 팀에만 두 달을 더 주는 식이죠. 전체를 미루면 이미 이전을 끝낸 팀이 손해를 보고, 다음번에는 아무도 서둘러 움직이지 않습니다.

다음으로 이전 작업을 상대에게 전부 떠넘기지 않습니다. 대체 코드 예시, 변환 스크립트, 첫 한 건의 시범 이전까지 해 주면 진행 속도가 크게 달라집니다. 폐기를 요청하는 쪽이 이전 비용의 일부를 부담한다는 원칙을 세워 두면 협상이 짧아집니다.

마지막으로 진행 현황을 공개합니다. 남은 호출자 목록과 각각의 상태를 모두가 보는 곳에 둡니다. 이름이 적힌 표는 독촉 메일 열 통보다 강합니다. 다만 목록의 목적이 비난이 아니라 조율이라는 점을 말과 형식 모두로 분명히 해야 합니다.

3.10 짧은 체크리스트

- 되돌리기 계획을 삭제 전에 썼는가
- 예고가 사용자가 실제로 보는 자리에 있는가
- 차단을 코드 삭제 없이 스위치로 할 수 있는가
- 소등을 짧은 것부터 점증했는가
- 제거를 되돌리기 쉬운 순서로 했는가
- 데이터를 마지막에 남겼는가
- 백업 복원을 실제로 해 봤는가
- 복구 창의 종료일이 카드에 적혀 있는가

여덟 개 중 하나라도 아니라면 아직 지울 때가 아닙니다.

4 데이터와 계약은 코드보다 늦게 죽는다

코드는 되돌릴 수 있습니다. 커밋을 되돌리고 배포하면 5분 안에 원래대로 돌아옵니다. 데이터는 다릅니다. 테이블을 드롭하면 백업이 있어야만 돌아오고, 백업 복원은 5분이 아니라 몇 시간짜리 작업입니다. 외부에 공개한 계약은 더합니다. 되돌리는 데 필요한 건 우리 배포 주기가 아니라 상대 회사의 일정이니까요.

그래서 앞 장의 제거 순서에서 데이터와 계약을 맨 뒤에 뒀습니다. 이 장은 그 마지막 구간을 다룹니다.

4.1 계약을 없애기 전에 호출자를 세는 법

공개 API, 웹훅, 이벤트 토픽, 공유 뷰는 우리 코드베이스 검색으로는 절대 판정할 수 없습니다. 호출자 식별이 먼저입니다.

식별 가능한 축을 순서대로 확인합니다.

- 인증 정보. 토큰, 키, 클라이언트 식별자별 호출량. 가장 정확합니다.
- 사용자 에이전트 문자열과 클라이언트 버전 헤더. 정확하진 않지만 소프트웨어 종류를 알려 줍니다.
- 출발지 주소 대역. 사내인지 특정 파트너인지 구분하는 데 씁니다.
- 계약서와 청구 기록. 기술 지표에 안 잡히는 사용자가 여기 남아 있는 경우가 있습니다.

식별이 안 되는 호출이 남으면 그 자체가 결함입니다. 이번 폐기가 아니라도 언젠가 필요하니, 인증 없이 열려 있는 경로에 최소한의 식별을 붙이는 작업을 함께 진행할 가치가 있습니다.

호출자 목록이 나오면 각각을 이전 완료, 이전 중, 연락 두절 세 칸으로

분류합니다. 마지막 칸이 핵심입니다. 연락이 안 되는 호출자에게는 마찰 단계가 사실상 유일한 소통 수단입니다. 한도를 낮추고 지연을 넣으면 대개 며칠 안에 연락이 옵니다.

4.2 API 종료 절차

버전이 있는 API라면 절차가 단순합니다. 새 버전을 내고, 구 버전에 종료일을 붙이고, 그날 끕니다. 문제는 대부분의 내부 API에 버전이 없다는 점이죠.

버전이 없어도 다음은 할 수 있습니다.

- 응답에 종료 안내 헤더를 넣습니다. 종료 날짜와 안내 문서 주소를 담습니다.
- 오류가 아닌 정상 응답에 함께 넣습니다. 오류에만 넣으면 아무도 못 봅니다.
- 종료일이 가까워지면 짧은 소등을 예고하고 실행합니다. 이때 반환하는 상태 코드와 본문을 미리 정해 두고, 그 내용에 안내 문서 주소를 넣습니다.
- 완전히 끈 뒤에도 최소 한 분기 동안은 410 같은 명시적 종료 응답을 유지합니다. 경로를 통째로 없애서 404가 나오면 상대는 자기 설정이 틀렸다고 오해하고 헤맵니다.

마지막 항목이 실무에서 가장 자주 빠집니다. 종료를 알리는 응답 자체가 한동안 살아 있어야 합니다. 그것도 폐기 계획의 일부입니다.

4.3 스키마 폐기 5단계

테이블이나 컬럼을 없앨 때는 다음 순서를 지킵니다. 각 단계 사이에 최소 한 번의 릴리스와 관측 기간을 둡니다.

단계	조치	되돌리기
1	쓰기 중단	코드 되돌림
2	읽기 중단	코드 되돌림
3	이름 변경	이름 복구
4	사본 보관	사본 복사
5	드롭	백업 복원

쓰기를 먼저 멈추는 이유는 그 순간부터 데이터가 더 이상 늘지 않기 때문입니다. 읽기를 먼저 끊으면 쓰기만 남아 아무도 안 보는 데이터가 계속 쌓입니다.

3단계 이름 변경이 이 절차의 핵심 장치입니다. 지우는 대신 이름을 바꾸면 남은 참조가 즉시 드러납니다. 예상치 못한 배치나 리포트가 있었다면 다음 실행에서 실패하고, 우리는 이름을 되돌려 몇 분 만에 복구합니다. 드롭했다면 같은 상황에서 백업 복원 작업이 필요했겠죠. 컬럼 이름 뒤에 폐기 예정 표시와 날짜를 붙이는 방식이 흔합니다.

4단계는 드롭 직전 사본을 만드는 것입니다. 별도 아카이브 공간이나 파일로 내보내 둡니다. 사본을 어디에 두는지, 언제까지 두는지, 누가 접근할 수 있는지를 카드에 적습니다. 개인정보가 포함된 데이터라면 사본을 만드는 행위 자체가 위험을 늘린다는 점도 함께 판단해야 합니다.

4.4 그림자 읽기와 격리 확인

이름 변경만으로 불안하다면 한 단계를 더 넣습니다. 대상 테이블 접근을 별도로 계측하는 것입니다. 데이터베이스에 따라 테이블 단위 접근 통계나 감사 기능을 제공하고, 없다면 전용 계정과 권한을 분리해 접근 시도를 로그로 남기는 방법도 있습니다.

권한 회수는 이름 변경보다 부드러운 격리 방식입니다. 애플리케이션

계정에서 해당 테이블의 조회 권한만 회수하면, 남은 참조가 명확한 권한 오류로 드러납니다. 되돌리기는 권한 한 줄을 다시 주는 것으로 끝나니 가장 싼 실험입니다.

반대로 새 구조로 옮겨 가는 중이라면 그림자 읽기를 씁니다. 새 경로와 옛 경로를 동시에 읽어 결과를 비교하고, 차이가 나면 기록만 남깁니다. 일정 기간 차이가 0에 수렴하면 옛 경로를 끕니다. 비용은 들지만 데이터 이관 사고를 막는 데는 이만한 방법이 없습니다.

4.5 지울 수 없는 데이터를 다루는 법

모든 데이터를 지울 수 있는 건 아닙니다. 세금, 회계, 의료, 금융 영역에는 법으로 정해진 보존 기간이 있습니다. 진행 중인 분쟁이나 감사와 관련된 자료도 마찬가지입니다. 여기서 성실한 기술적 판단이 법적 문제로 바뀝니다.

원칙은 간단합니다. 확인 없이 지우지 않고, 확인 없이 남기지도 않습니다. 두 방향 모두 위험합니다. 필요 이상으로 오래 보관한 개인정보 역시 감사에서 지적받는 항목이니까요.

지울 수 없다고 판정된 데이터는 다음 중 하나로 처리합니다.

- 동결. 쓰기와 읽기를 모두 끊고 접근 권한을 최소 인원으로 줄입니다. 애플리케이션에서는 완전히 사라진 것과 같은 상태가 됩니다.
- 이관. 운영 데이터베이스에서 저비용 보관소로 옮깁니다. 성능 부담과 사고 표면이 함께 줄어듭니다.
- 축약. 보존 의무가 있는 항목만 남기고 나머지 컬럼은 지웁니다. 전체 행을 남길 필요가 없는 경우가 많습니다.
- 소유권 이전. 이 데이터를 필요로 하는 팀이 따로 있다면 그 팀의 저장소로 넘기고 우리 코드에서는 끊습니다.

네 가지 모두 코드 관점에서는 삭제와 같은 효과를 냅니다. 우리 시스템이

더 이상 이 데이터를 알지 못하게 된다는 점에서요. 폐기의 목표가 물리적 소멸이 아니라 유지 책임의 종료라는 걸 기억하면 선택지가 넓어집니다.

4.6 이벤트와 큐는 조용히 살아남습니다

메시지 큐와 이벤트 스트림은 폐기 목록에서 자주 빠집니다. 화면처럼 눈에 보이지 않고, API처럼 응답 코드로 신호를 줄 수도 없기 때문입니다. 그러면서 저장 공간과 운영 부담은 계속 씁니다.

여기서는 세 가지를 확인합니다. 이 토픽에 아직 쓰는 쪽이 있는지, 읽는 쪽이 있는지, 읽는 쪽이 실제로 그 내용으로 무언가를 하는지입니다. 세 번째가 핵심입니다. 소비자가 붙어 있지만 받은 메시지를 즉시 버리는 경우가 놀랄 만큼 흔합니다. 소비자 지연이 0인데 처리 결과 지표가 없다면 의심할 만한 신호입니다.

폐기 순서도 방향이 반대입니다. 여기서는 생산을 먼저 멈춥니다. 생산이 멈추면 소비자는 자연히 조용해지고, 남은 메시지가 처리된 뒤 토픽을 지웁니다. 소비자를 먼저 없애면 메시지가 쌓여 저장소 경보가 울립니다.

4.7 삭제 이후를 문서로 남기기

폐기가 끝나면 코드에서 흔적이 사라집니다. 그래서 반대로 문서가 필요합니다. 나중에 누군가 “예전에 이런 기능이 있었다는데 왜 없어졌나요”라고 물을 때 답할 근거죠.

한 페이지면 충분합니다. 무엇이 있었고, 언제 왜 없앴고, 대신 무엇을 쓰며, 데이터는 어디에 있는지. 이 기록이 없으면 몇 년 뒤에 같은 기능이 다시 만들어집니다. 폐기의 반대편 실패는 부활입니다.

4.8 이름도 부채입니다

마지막으로 자주 놓치는 항목이 있습니다. 이름입니다. 폐기된 기능의 이름이 코드, 문서, 대시보드, 회의록에 남아 있으면 몇 년 뒤 새로 온 사람이 그 이름을 검색하고 혼란에 빠집니다. 정산v1을 지웠다면 정산v2를 그냥 정산으로 되돌리는 작업까지가 한 세트입니다.

이름 정리는 급하지 않지만 미루면 다음 세대의 판정 비용을 높입니다. 폐기 카드를 닫기 전에 남은 이름을 한 번 검색해 보세요. 대개 열 군데 남짓이고, 30분이면 정리됩니다.

5 폐기를 상시 습관으로 만드는 운영

여기까지의 내용을 잘 따르면 기능 하나는 깨끗하게 사라집니다. 그런데 다음 분기에 새 기능 열두 개가 들어오고, 그중 넷은 아무도 안 쓰게 됩니다. 한 번의 대청소로는 이길 수 없는 게임입니다.

마지막 장은 개별 폐기가 아니라 폐기율을 다룹니다. 만드는 속도보다 정리하는 속도가 느리면 부채는 계속 늘어납니다. 목표는 두 속도를 비슷하게 맞추는 것입니다.

5.1 만료일을 기본값으로

가장 효과가 큰 습관 하나만 고르라면 이겁니다. 새로 만드는 것에 처음부터 종료 조건을 붙이는 습관이죠.

- 실험용 기능 플래그에는 만료일을 필수로 넣습니다. 만료일이 지나면 파이프라인이 경고하거나 실패합니다.
- 임시 우회 코드에는 제거 조건을 주석이 아니라 이슈로 남깁니다. 주석은 아무도 검색하지 않습니다.
- 새 API에는 처음부터 버전을 붙입니다. 버전 없는 계약은 나중에 종료 절차가 없습니다.
- 파일럿과 시범 서비스에는 종료일과 함께 “이 조건을 못 채우면 접는다”를 문서에 적습니다.

핵심은 나중에 판단하겠다는 약속이 아니라, 아무도 손대지 않으면 자동으로 사라지는 구조입니다. 사람의 의지에 기대는 정리는 반드시 밀립니다.

플래그 수명은 특히 잘 드러나는 지표입니다. 열려 있는 플래그 수가 계속 늘고 있다면 이미 문제가 시작된 겁니다. 실험은 끝났는데 플래그만 영원히

남아 분기 두 갈래를 다 유지하는 상태가 조용히 쌓입니다.

5.2 삭제 예산

폐기 작업이 우선순위 회의에서 이기게 만들려고 애쓰지 마세요. 애초에 경쟁시키지 않는 편이 낫습니다. 별도 예산을 떼어 둡니다.

운영 방식은 팀에 맞게 고르면 됩니다.

- 스프린트 용량의 일정 비율을 폐기에 고정 배정합니다. 10퍼센트 정도면 눈에 띄는 변화를 만듭니다 [추정].
- 분기마다 폐기 목표를 두세 개로 정하고 다른 목표와 같은 지위로 관리합니다.
- 큰 기능을 새로 만들 때 관련된 폐기 후보 하나를 함께 계획에 넣습니다. 새것과 옛것을 같은 작업으로 묶으면 이전이 훨씬 자연스럽습니다.

세 번째가 가장 실용적입니다. 정리만 따로 하는 시간은 지루하고 정치적으로 약하지만, 새 기능을 만들며 옛 경로를 걷어내는 일은 그 작업의 일부처럼 보입니다. 실제로도 그렇고요.

반대로 피할 방식도 있습니다. 일 년에 한 번 잡는 대청소 주간입니다. 하루 이틀 안에 판정과 예고와 소등을 다 하려니 절차가 무너지고, 사고가 나면 다시는 그 행사를 못 하게 됩니다.

5.3 폐기 백로그와 30분 회의

후보를 모아 두는 자리가 필요합니다. 이슈 목록에 라벨 하나면 충분하고, 도구는 중요하지 않습니다. 중요한 건 항목마다 다음 세 가지가 적혀 있다는 점입니다. 현재 증거 등급, 다음에 필요한 조치, 소유자.

한 달에 한 번 30분 회의를 잡습니다. 안건은 셋입니다.

- 지난달에 닫힌 카드 확인
- 진행 중인 카드의 등급이 올랐는지 확인
- 새 후보 한두 개 등록

이 회의는 논쟁하는 자리가 아닙니다. 증거 등급이 조치를 결정하니 대부분은 5분 안에 정리됩니다. 회의가 길어진다면 그건 등급 규칙이 아직 팀 합의가 아니라는 신호입니다.

5.4 추적할 지표 넷

관리하고 싶다면 재야 합니다. 다만 지표를 많이 만들면 지표 자체가 부채가 되니 넷 정도로 제한합니다.

지표	보는 이유
열린 플래그 수명	부채 유입 속도
미사용 표면 개수	정리 대상 규모
폐기 리드타임	절차의 마찰
삭제 대 추가 비율	방향 균형

플래그 수명은 중앙값으로 봅니다. 평균은 오래된 하나에 끌려갑니다. 미사용 표면은 증거 등급 E1 이상인 항목 수로 세면 계산이 쉽습니다. 폐기 리드타임은 카드가 열린 날부터 닫힌 날까지의 기간입니다. 이 값이 길다면 절차 어딘가에 병목이 있고, 대개 예고 승인 아니면 다른 팀 이전 대기입니다.

마지막 비율은 해석에 주의가 필요합니다. 삭제 줄 수를 성과로 걸면 사람들이 줄 수를 조작합니다. 목표가 아니라 방향 확인용 신호로만 쓰세요.

5.5 무해한 실패 문화

1장에서 삭제가 밀리는 이유로 책임 비대칭을 들었습니다. 이 문제는 지표로 풀리지 않고 규칙으로 풀립니다.

절차를 지켜서 지웠는데 문제가 생겼다면 그건 개인의 잘못이 아니라 절차의 결함입니다. 사후 검토에서 물어야 할 질문은 "누가 지웠나"가 아니라 "어떤 증거가 없었길래 판정이 틀렸나"입니다. 그 답이 다음 카드의 체크리스트 한 줄이 됩니다.

같은 이유로 복구를 축하할 필요가 있습니다. 소등 중에 문제를 발견하고 3분 만에 되돌렸다면 그건 성공한 작업입니다. 되돌린 횟수를 실패 건수로 세는 순간 아무도 소등 훈련을 하지 않게 되고, 결국 검증 없는 삭제만 남습니다.

5.6 30일, 60일, 90일

처음 시작하는 팀을 위한 순서를 적어 둡니다.

첫 30일에는 목록과 규칙만 만듭니다. 후보 열 개 남짓을 모으고, 1장의 원가 추정으로 순위를 매기고, 증거 등급 표를 팀 문서에 올립니다. 아직 아무것도 지우지 않습니다. 그리고 상위 후보 셋에 계측을 붙입니다.

다음 30일에는 가장 쉬운 하나를 끝까지 통과시킵니다. 원가가 큰 것 말고, 화석에 가깝고 되돌리기 쉬운 것을 고르세요. 목적은 절차를 한 바퀴 돌려 보는 것입니다. 예고문 양식, 소등 절차, 복구 훈련, 카드 닫기까지 실제로 해 보면 팀 안에 경험이 생깁니다.

마지막 30일에는 습관을 심습니다. 플래그 만료 규칙을 파이프라인에 넣고, 월 1회 회의를 캘린더에 고정하고, 지표 넷을 대시보드에 올립니다. 그리고 진짜 비싼 후보에 착수합니다. 이때쯤이면 팀이 절차를 신뢰하니 큰 대상에도 손이 갑니다.

5.7 만들기 전에 지울 계획을 세운다

가장 값싼 폐기는 애초에 만들지 않는 것입니다. 그다음으로 싼 것은 만들 때 끝을 정해 두는 것이고요.

새 기능 기획서에 한 문단을 추가해 보세요. 제목은 “이 기능을 언제 어떻게 접는가”입니다. 여기에 세 줄만 적습니다. 성공 기준과 관측 방법, 실패했을 때의 종료 조건, 그리고 종료 시 남는 데이터의 처리 방향.

이 문단을 요구하면 재미있는 일이 벌어집니다. 몇몇 기획이 작성 단계에서 조용히 취소됩니다. 성공 기준을 못 적는다는 건 무엇을 기대하는지 모른다는 뜻이니까요. 통과한 기획도 나중에 판정하기가 훨씬 쉬워집니다. 6개월 뒤 폐기 회의에서 “원래 기준이 이거였는데 못 넘었네요”라는 문장 하나로 논의가 끝납니다.

기술 부채를 감정 없이 다루는 방법이 여기 있습니다. 미리 정한 기준으로 판정하면 그건 누구의 실패도 아니고 그냥 실험 결과입니다.

5.8 계약직 코드라는 개념

팀에 도움이 되는 표현 하나를 소개합니다. 코드를 정규직과 계약직으로 나누는 방식입니다.

정규직 코드는 제품의 핵심이고, 장기 유지를 전제로 설계하며, 문서와 테스트에 투자합니다. 계약직 코드는 특정 목적과 기간을 위해 만들고, 그 목적이 끝나면 나갑니다. 마이그레이션 스크립트, 캠페인용 화면, 임시 연동, 실험 분기가 여기 속합니다.

문제는 계약직으로 들어온 코드가 계약 종료일 없이 눌러앉는 것입니다. 그래서 만들 때 어느 쪽인지 표시합니다. 저장소 위치를 나누거나, 이름 규칙을 정하거나, 소유자 파일에 종료일을 적는 식이면 됩니다. 방법은 자유롭고 구분 자체가 목적입니다.

이 구분이 자리 잡으면 폐기 회의의 절반이 사라집니다. 계약직 코드는 논쟁 없이 기간 만료로 처리되니까요.

5.9 마지막으로

폐기는 화려한 작업이 아닙니다. 잘하면 아무 일도 일어나지 않고, 사용자는 알아채지 못하며, 릴리스 노트는 짧습니다. 남는 것은 다음 분기에 조금 더 빨라진 배포와 조금 덜 무거운 온보딩입니다.

그 조용함이 이 규율의 성과 지표입니다. 무언가를 없앤 날 아무 일도 없었다면, 절차가 제대로 작동한 겁니다.